SHAMROCK: A Synthesizable High Assurance Cryptography and Key Management Coprocessor

David Whelihan, Michael Vai, Dan Utin, Roger Khazan, Karen Gettings, Tom Anderson,
Antonio Godfrey, Raymond Govotski, Mark Yeager, Brendon Chetwynd, Ben Nahill, and Eric Koziel
Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, MA 02420

***Abstract* — For performance, maintainability and usability, military communications systems must properly integrate and coordinate cryptographic primitives and use adequate key management schemes. In this paper, we present a SHAMROCK (Synthesizable High Assurance Management/Reservation/ Operation of Cryptography and Keys) coprocessor. Being self-contained and synthesizable, SHAMROCK empowers designers to readily and correctly incorporate cryptography and key management into embedded systems. SHAMROCK has been incorporated in multiple mission critical systems to enable secure computing and communications.**

Keywords — secure communications; security coprocessor; high assurance; cryptography; key management; synthesizable.

I. INTRODUCTION

Modern cryptographic algorithms are designed with Kerckhoffs' Principle in mind – that a system should remain secure even when the attacker knows everything from algorithms to code structure, as long as the secret key is protected [1]. The security of such a system hinges upon the effectiveness of key management, which is the process and apparatus for creating, disseminating, using, and revoking cryptographic keys.

Unlike standard cryptographic operations (e.g., encryption), key management needs to be customized according to the nature of an application [2]. Any hardware/software vulnerabilities in key management will result in the compromise of communication security.

This paper describes the architecture and implementation details of SHAMROCK, which is a synthesizable coprocessor for cryptography and key management developed to simplify the creation of robust secure systems. SHAMROCK is a hardware implementation of the Lincoln Laboratory Open Cryptographic and Key Management Architecture (LOCKMA) software library package, which enables rapid dynamic rekeying of communicating devices in real-time.

LOCKMA supports good cryptography and key management practices. However, in general a software only solution is insufficient to guarantee the security of the keys as system software (e.g., the operating system) could have unrestricted access to the memory used for processing keys. Special hardware extensions are necessary to achieve high assurance that keys are kept secret. SHAMROCK addresses high assurance by physically separating critical information from its host application.

The paper is structured as follows: Section 2 briefly reviews Public Key Infrastructure (PKI) based key management technology. Section 3 describes LOCKMA's dynamic, distributed key management in detail. Section 4 discusses the implementation of the high assurance SHAMROCK coprocessor.

II. KEY MANAGEMENT BACKGROUND

Distributed, dynamic systems are connected via complex, real-time communication networks. The essential requirement of such systems is their resiliency to unintentional (e.g., bugs) and maliciously induced (e.g., attacks) faults. Successful operation is enabled by allowing individual devices to communicate with each other over encrypted channels and dynamically including and excluding devices from a communicating group as they become trusted, or untrusted.

An example of using rekeying to dynamically adjust group membership for three devices, A, B, and C, is illustrated in Fig. 1. These devices communicate to each other via encrypted channels. In Fig. 1a, the three devices are able to communicate using the same key. If device C is no longer used (or trusted) for the application, the system is reconfigured by excluding device C from the communicating group. This process is described in Fig. 1b, where device A initiates a rekey operation by sending out a keywrap that can be unwrapped only by device B. As shown in Fig. 1c, the encrypted communication now switches to the new key contained in the keywrap, thus excluding device C from the communicating group.

The critical technology that enables such rekeying is PKI [3]. In the commonly used key agreement function, such as Elliptic Curve Diffie-Hellman (ECDH), each communicating device holds a secret private key and an associated public key. Communicating devices can exchange their public keys over unprotected channel and use ECDH to derive a shared secret. The shared secret is then used to establish symmetric keys for the encrypted communication channels.

A digital signature is an important function in PKI. Digital signature algorithms, such as the Elliptic Curve Digital Signature Algorithm (ECDSA), work by computing a cryptographic hash of a message using, for example, the Secure Hash Algorithm (SHA). The message is then signed by deriving a value based on a private key, the message hash, and other meta-data. The receiver of a message can verify its source by running ECDSA using the message hash and the matching public key.

Distribution A: Public Release. This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721- 05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

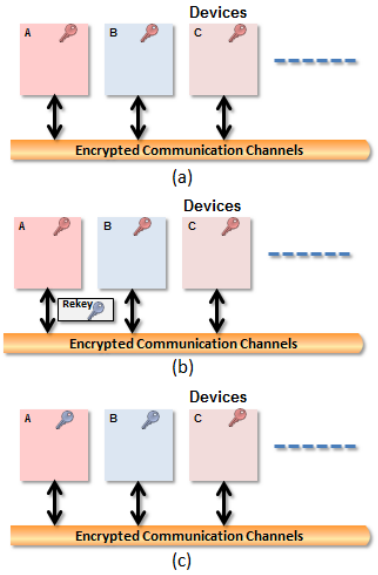

Fig. 1. Group management enabled by dynamic rekeying.

Another critical aspect of PKI is the Certificate Authority (CA), which is an entity that issues digital certificates to certify the ownership of a public key by the named subject of the certificate. Using the CA's public certificate, one can verify that an issued certificate was signed with the CA's private key. If the device trusts the CA and verifies that it has received a CA signed certificate, then the device can trust the certificate.

III. LOCKMA Library

LOCKMA is a highly portable, modular, open software library of key management and cryptography algorithms, which provides user, identity, and key management functions, as well as support for hardware and software Suite B cryptographic primitives [4]. LOCKMA has an intuitive front-end Application Programming Interface (API) so developers can easily create security functions.

Fig. 2 shows LOCKMA's interfaces as high-level security functions and low-level cryptographic primitives. Complicated cryptographic algorithms are captured as core modules. The back-end API supports the use of low-level cryptographic kernels implemented in either hardware or software.

LOCKMA uses PKI, along with cryptographic functions such as Advanced Encryption Standard (AES) [5] to produce keywraps to orchestrate key exchanges between groups of devices. These key exchanges are facilitated by the generation of multi-recipient *keywraps* that encrypt a set of symmetric keys into a Key Update Block (KUB). The structure of a keywrap generated by the LOCKMA library is shown in Fig. 3.

Fig. 3 illustrates the keywrap's nested structure with the outer layer consisting of the certificate of the keywrap sender, and a signed KUB. Each device that runs LOCKMA must have at least one public-private key pair for key agreement. The keywrap issuer must have an additional key pair for signing of the KUB. The issuer of a keywrap must include its public signing key in a certificate signed by an authority trusted by all participants in the communicating group.

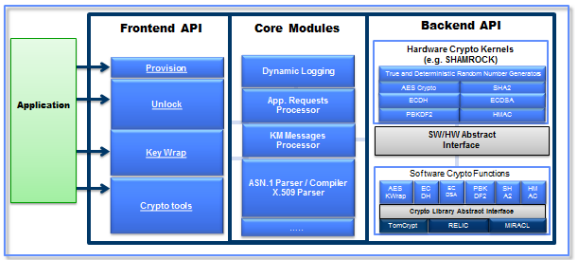

Fig. 2. LOCKMA provides application interface, high level security function, and low level cryptographic kernels.

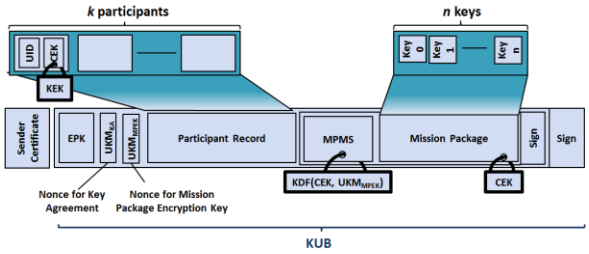

Fig. 3. LOCKMA keywrap structure.

The signed KUB has four components: the issuer's Ephemeral Public Key (EPK) used for key agreement; two one-time single-use Numbers-Used-Once, or *nonces* UKM_{KA} (for key agreement) and UKM_{MPEK} (for mission package encryption key) that introduce unique randomness to subsequent transactions; a participant record that is used by authorized recipients to decrypt the cryptographic keys and associated meta-data in the keywrap; and the mission package, which contains the symmetric keys to be distributed. The participant record is a plaintext data structure that holds a set of data sub-structures, one for each authorized recipient, that in turn contain encrypted copies of a Content Encryption Key (CEK) that is used to encrypt the mission package. The CEK is replicated once for each participant, and encrypted with a Key Encryption Key (KEK) derived using ECDH as follows:

$$KEK_i = ECDH(k_{pri\text{-}e}, k_{pub\text{-}i}, UKM_{KA}),$$

where $k_{pub\text{-}i}$ is the i^{th} recipient's public key agreement key. $K_{pri\text{-}e}$ is the keywrap issuer's ephemeral private key agreement key generated only for the present keywrap. Ephemeral key agreement key is required because ECDH with two static keys is not approved for key agreement by NIST [6].

The mission package contains an arbitrary number of symmetric keys that are to be used by the authorized recipients for the encryption of data. The meta-data that describes what the keys are used for, such as key expiration times and algorithm specifiers is stored separately in the Mission Package Metadata Storage (MPMS). This is done for two reasons: first, the mission package is encrypted using AES keywrap mode with the CEK [7], which is a special version of AES that is designed to safeguard only keys, and second, there are information assurance implications to mixing secret keys and metadata. While not strictly secret, key meta-data can leak information about the application that will use the keys. To further enforce separation, the key metadata is also encrypted but with an irreversible Key Derivation Function (KDF) using the CEK and the second nonce UKM_{MPEK}. The combination of

the metadata and mission key set is collectively signed by the device issuing the keywrap.

LOCKMA implements cryptographic functions via the RELIC [8] and TomCrypt [9] open source toolkits. It parses certificates using an X.509 certificate parser [10]. Keywrap messages are formatted using the Cryptographic Message Syntax (CMS) [11], and parsed using an ASN.1 syntax parser [12].

Being a software library, LOCKMA can provide robust and flexible key management functionality, but it cannot guarantee that generated secret keys are not leaked. This is because LOCKMA's ability to enforce security policies – is limited by the degree of assurance of the application it is linked into. In the next section a high-assurance version of LOCKMA implemented in hardware, whose goal is to allow software to use keys but never possess keys, is described.

IV. SHAMROCK COPROCESSOR

The overarching goal of the SHAMROCK processor is to provide the static cryptographic operations necessary to create secure cryptographic channels between communicating devices without hampering the system designer's ability to conform to arbitrary communication protocols and system architectures.

High-assurance is an important goal of many military systems. Generally, when we use the term assurance, we are speaking of assurance that established security policies (such as the protection of secret key material) are being obeyed by the system under a variety of operating conditions. In the case of cryptographic processing, these policies are relatively straightforward:

1. The value of secret keys used to encrypt data or other keys must never leak out.
2. The value of critical secret data must never leak out.

Here, we differentiate between data and keys, as a leak of key material can greatly amplify data loss. Keys are used in a limited context – for example, as an input to a streaming encryption/decryption function – and are easier to protect than user data, which is used (in unencrypted form) in a potentially unlimited set of contexts. Further, those contexts frequently have usability constraints placed on them that limit applicable data security techniques. Therefore, any assurance arguments made for LOCKMA, or its hardware implementation described below can only make claims on policy 1. However, the ease of deployment of LOCKMA or SHAMROCK in secure systems has lessened the burden on a system designer in meeting policy 2 as effective key management is easier to achieve and use in order to adequately protect data, as the effective key management is essential to effective data protection.

The primary goal of the SHAMROCK processor is to encapsulate the LOCKMA key management system in a high-assurance "shell" that is as agnostic to protocol as possible. This way the nuances of application specific encoding methods such as CMS, X.509 certificate parsing, and Ethernet can be separated from the core functionality of generating and distributing keys and providing cryptographic services. This separation is very much akin to the goals of the LOCKMA software library, but in a physical form and with a focus on high-assurance and separation of critical cryptographic functionality from application functionality. The primary methods for achieving high-assurance are:

- Rigid and architecturally enforced separation of data and key material into concentric security zones.
- Minimized and rigidly defined external interfaces.
- Physically verifiable inter-zone interfaces.

Using these techniques which will be described in following sections, SHAMROCK achieves an important goal: Complex systems that implement functionality in software can use cryptographic functions whose security relies on secret key material without ever actually possessing that key material.

A. Threat Model

SHAMROCK is a synthesizable coprocessor that is written in a Hardware Description Language (HDL) and can therefore be implemented in an Application-Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). Salient assumptions of the operating environment are:

- Malicious actors may obtain physical control of some SHAMROCK-enabled devices, and will be able to query and attempt to utilize SHAMROCK for cryptographic operations.
- SHAMROCK and a separate non-volatile memory (NVRAM) used for long-term storage of credentials are inside of a physical protected volume. The FPGA device for prototyping does not have on-chip NVRAM, so the architecture includes an NVRAM Interface.
- Mitigation against side-channel attacks, such as Differential Power Analysis (DPA) [13], will be provided separately from the SHAMROCK core. The SHAMROCK architecture does not preclude the inclusion of side-channel-resistant crypto-cores, but the system described in this paper does not directly address this type of attack.
- The core is implemented correctly, and without any added nefarious logic.

B. High Level Architecture

Fig. 4 shows the high level architecture boundary of SHAMROCK. SHAMROCK has three major functions: key protection and manipulation, streaming symmetric encryption/decryption, and key management protocol. All three of these functions are architecturally separated from one another, and communicate via simple interfaces that are designed to minimize attack surface. The SHAMROCK prototype currently implements an AES-CTR (counter) mode encryptor/decryptor [14]. However, this encryptor/decryptor can be replaced with any user-supplied symmetric cipher block.

The key management functionality is divided up into an untrusted protocol control module, which hereafter is referred to as the *black module*, and a monolithic high-assurance key management (HAKM) module that implements key derivation, key agreement, and key protection, which is called the *red module*.

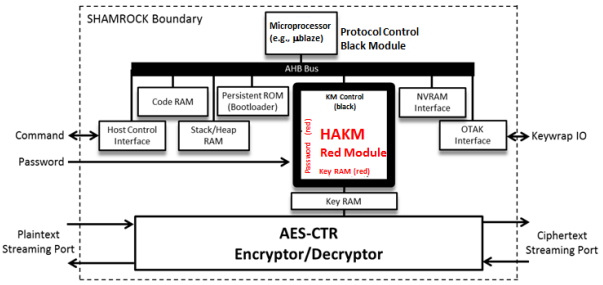

Fig. 4. SHAMROCK high level architecture.

The black module is implemented with a small "soft" microprocessor (e.g., the Xilinx MicroBlaze) that is responsible for external communication as well as parsing and generation of cryptographic messages. The microprocessor is the master on an AMBA High-performance Bus (AHB) [15] and drives a number of connected devices including I/O transceivers, RAM, ROM and the HAKM red module.

The black module microprocessor boots from a special trusted persistent ROM (bootloader) and uses the HAKM red module to verify its run-time code image, which is digitally signed by a trusted third party. External interfaces generally obey a simple First-In-First-Out (FIFO) protocol for maximum flexibility. Those interfaces are described below:

Password (red): SHAMROCK's root of trust and root credentials are persistent across power cycles. These data are "unlocked" by supplying a password to a special red port at power-on time.

Command (black): A command-line based interface to SHAMROCK's key management subsystem.

Keywrap IO (black): An external-facing port used to receive and transmit keywraps into the Over-The-Air-Keying (OTAK) interface.

Plaintext streaming port (red): The red interface to the streaming symmetric encryptor/decryptor.

Ciphertext streaming port (black): The black interface to the streaming symmetric encryptor/decryptor.

With the exception of the password port, none of these interfaces directly connects to the HAKM red module. Rather, the protocol control black module handles all input and output, and only passes vetted data (primarily keywraps) through. Except for the key RAM output, which provides received keys for the streaming encryptor/decryptor, the HAKM red module does not have connections through which keys can be divulged.

Similarly, the HAKM red module has a minimal interface to the rest of the SHAMROCK system. Its ports are described below:

KM control (black): A command/status interface used to pass data and commands to the HAKM red module.

Password (red) – An input-only port used to receive passwords from outside the device.

Key RAM (red) – An output port that serves received keys out to the streaming encryptor/decryptor.

Given the assumption that the protocol control black module is untrusted, the HAKM red module thus implements a concentric security architecture that utilizes cryptographic functions to control information flow and enable security guarantees.

C. High Assurance Key Management Architecture

The design goal is to ensure that a compromised protocol control module cannot extract root secrets, or secrets generated in another SHAMROCK device from the local HAKM red module. The HAKM red module design therefore assumes that it is being used/controlled by a competent external entity, but it does not trust that entity to manage or control secret key material.

The HAKM red module security is strongest when it is communicating keywraps to/from other HAKM modules (as opposed to LOCKMA software instances, which can also generate and unwrap keywraps). This is due to a founding principle of the HAKM module architecture; secret material is generated deep inside of the HAKM module and can only be exported by modifying (e.g., encrypting or hashing) it using other data/key material generated inside of an HAKM module. Thus, keys generated inside of the HAKM module can only exist in plaintext form when they are inside of a trusted HAKM module, with the exception of keys that are supplied to the encryptor/decryptor. This is underpinned by the "logical unknowability" of data in key generation regions of the HAKM module.

This design principle is illustrated in Fig. 5, in which two SHAMROCK devices are communicating over a wireless channel. The concentric rectangles represent the security zones in the HAKM module, with the inner area (Zone 0) containing root secrets generated inside of the HAKM module, and the next region (Zone 1) denoting ephemeral secrets such as dynamic keys. AES encryption is employed as a one-way function leading from the ephemeral Zone 1 to the outermost Zone 2, which could be considered outside of the high-assurance boundary. While the root secret (represented as the inner-most key in Zone 0, Fig. 5) is used to transform the ephemeral secret (represented as the key in Zone 1, Fig. 5) using AES, the root secret itself never leaves the confines of Zone 0. As the inputs to the AES unit can only come from Zones 0 and 1, then the secret is protected by encryption outside of the HAKM module (i.e., Zone 2 and beyond).

This high assurance architecture is made possible by several combined technologies and techniques, including in-situ key generation, and physically directed cryptographic functions. Directed cryptographic functions are functions such as AES or ECDH with their physical implementation laid out so that they can only be used to transfer and transform data from one physical location to another. Their layout in the system is therefore tightly linked to the aggregate functions in which they will be used.

As an example, in the key management scheme laid out in Section III, an intermediate step in the keywrap generation is to

encrypt a set of keys with AES keywrap mode using a CEK. The plaintext keys to be encrypted reside in an ephemeral but secret Zone 1 memory, and will be encrypted and sent out to a Zone 2 black memory using a key that also resides in the Zone 1 memory. This operation could be carried out by a general-purpose AES engine that has read and write access to either memory, but such an arrangement would be vulnerable to misuse of the AES unit, either by copying plaintext keys across memories or encrypting with keys known to the adversary. In contrast, dedicated physically directed AES units are used, which can only output to a designated location, and take inputs (keys and data) from another separate designated location.

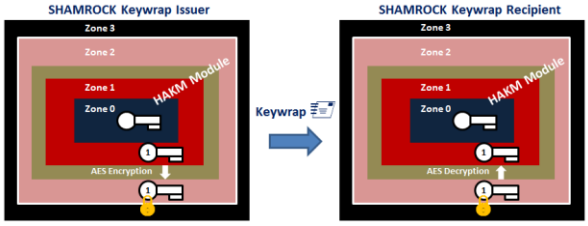

Fig. 5. HAKM module concentric secure zones.

Similarly, since the keywrap process requires unwrapping, there must be a reciprocal AES function that decrypts keys into the Zone 1 memory. Decryption from the black Zone 2 memory into the red Zone 1 memory can only be done using a key that already resides in the red Zone 1 memory. Since the contents of the key package are secret, decrypting ciphertext from the black Zone 2 memory into the red Zone 1 memory using a key in the red Zone 1 memory will result in secret data in the red Zone 1 memory, which is unknowable to the outside of Zone 1. This complies with the security tenet that all information originated from another HAKM module residing in the red Zone 1 memory will be secret and unknowable.

In the SHAMROCK design, the AES units used for high assurance purposes are all physically separate units, whose inputs are rigorously and physically verified and controlled during the design process. Fig. 6 shows the internal layout of the SHAMROCK HAKM module. From the outside, the module appears to have a contiguous memory space. In reality the HAKM module memory is broken up into the three zones illustrated in Fig. 5, with the memory in the outer SHAMROCK protocol control region considered a fourth. These zones are defined as follows:

Zone 0: Root keys and other long-term credentials. This region is rigorously controlled. Its contents are used for a limited set of operations, such as signing and key agreement.

Zone 1: Ephemeral secret information. Ephemeral keys for key agreement, as well as keys sent and received in keywraps. Content Encryption and Key Encryption Keys are generated and temporarily stored here.

Zone 2: Black staging. A non-secret memory that is used to stage and marshal data during long atomic operations such as provisioning or keywrap generation.

Zone 3: Black protocol. This memory is outside of the HAKM module, which is not uniform and contiguously addressed with respect to the HAKM memory.

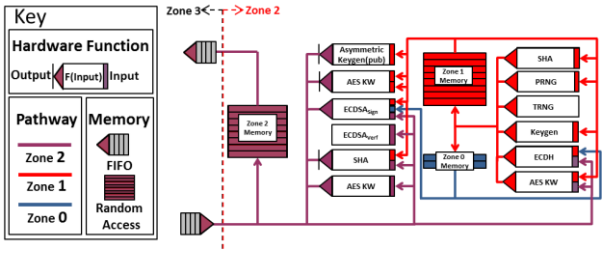

Fig. 6. SHAMROCK HAKM internal structure.

The interface to the HAKM module is via input and output FIFO memories and a special command mailbox register, each of which is individually addressable on the external AHB bus by the protocol control module. Input data can be written into the FIFOs, then a write command executed via the mailbox. On command execution, the HAKM takes control and copies that data into the specified location in Zone 2 memory. Attempting to copy data into any other zone by specifying an address that is mapped outside of the Zone 2 memory will result in an error because there is no direct path available.

D. SHAMROCK Operations

The following description is given at high level with respect to the current FPGA prototype of SHAMROCK, but is sufficient for understanding its operations in general. An example of the constraints placed by the selected FPGA is its lack of on-chip non-volatile memory, thus requiring the use of an external NVRAM.

In normal operation, the device is controlled by the software running on the protocol control module in Zone 3, which also handles all external interaction. Booting the protocol control module is thus the first step in SHAMROCK operation. SHAMROCK has an internal bootloader embedded into the FPGA bitstream, which is protected with the FPGA's built-in security features. The device relies on an external NVRAM to store an encrypted boot image. The role of the bootloader is to read a software image from the NVRAM, verify its signature for authentication, decrypt, and then start the execution. If the verification fails, the device halts.

When SHAMROCK is first powered on, it has no secret credentials and therefore no key management capability. It must be provisioned with at least one private and public key pair for key agreement purpose. In the case of a keywrap issuer, the device must also be provisioned with a pair of signing private and public keys. The secret portion of these key pairs is generated using the ring oscillator based true random number generator (TRNG) to seed a pseudo-random number generator (PRNG) (see Fig. 6). The storage of these long term key pairs is encrypted with a key generated by a user password. In addition, the entire provisioning function is designed to be atomic, which means that a single command triggers private key generation, password entry, and encrypted storage. These important keys are stored in Zone 0, the long-term secret store, which can only be written when the device is powered on. The use of these keys is then physically restricted to only certain Zone 1 functions.

After the SHAMROCK device is fully provisioned, it must be unlocked before use. This atomic operation involves

receiving a password entered through the password port and decrypting the long term credentials. The public key(s) and a couple of nonces (UKM_{KA} and UKM_{MPEK}) generated with the PRNG are placed in Zone 3. The device is now ready to process or, if it has a signing key, generate keywraps.

The first step in keywrap construction is the generation of one or more symmetric mission keys for encrypted communications between SHAMROCK-embedded systems. The key generation request is initiated by the protocol control module to the HAKM module, which creates and places them in Zone 1 memory. Note that the protocol control module knows of the existence of these keys, but has no ability to understand them. The protocol control module then generates metadata associated with each key, such as its creation time, expiration time, etc., and places them in Zone 2.

The protocol control module then requests the HAKM module to generate a CEK and encrypt the mission key package with a gate keeper AES encryptor going from Zone 1 to Zone 2. For the reason explained earlier in Section III, a derivation of the CEK is used to encrypt the metadata already in Zone 2.

Next the combined metadata and mission key package is signed and ready for insertion in the final keywrap structure. Next, participant records, one for each member of the communicating group, are created. The protocol control module directs the HAKM module to produce in Zone 1 a KEK for each intended participant using the participant's public key agreement key. The CEK is encrypted multiple times, one for each participant with its respective KEK. Associated participant identification tags (UIDs) are added to complete the participant record. The keywrap is completed by adding the sender's certificate, EPK, and the nonces. The now fully formed keywrap is signed before it is sent out.

When a keywrap is received, the recipient unwraps it to extract the current mission key set. This process is essentially the reverse of the above keywrap generation process. The only exception is that only the participant record associated with the recipient is decrypted with its KEK to retrieve the CEK.

E. Prototype Implementation and Evaluation

The SHAMROCK coprocessor has been prototyped on a Xilinx Kintex 7 FPGA-based testbed (Fig. 7). We have measured the performance of two main key management operations in the SHAMROCK prototype (with a 50 MHz clock). In a rekeying operation, the timing of generating and receiving keywraps on the FPGA are 337 ms and 130 ms, respectively. In a typical usage scenario, we assume that the streaming AES is used 100% of the time while the key management functionality is used 5% of the operational time. The average dynamic power for this scenario is ~53 mW. In particular, the key wrapping and key unwrapping operations consume ~23 mw and ~ 141 mw, respectively.

V. Summary and Ongoing Work

In this paper we have described a self-sufficient synthesizable high assurance cryptography and key management coprocessor. Comparing to its software counterpart, the hardware processor provides additional security and performance through a high assurance architecture and design. Ongoing work includes the development of a reference secure embedded system architecture for bringing dynamic rekeying to the secure computing and communications of distributed embedded systems [16].

Fig. 7. SHAMROCK testbed.

References

[1] Kahn D., The codebreakers: the comprehensive history of secret communication from ancient times to the Internet, Scribner, New York, 1996.

[2] Khazan, R., Figueriredo, R., McLain, C., Cunningham, R., Securing Communication of Dynamic Groups in Dynamic Network-Centric Environments, MILCOM 2006, Washington, DC, 23 October 2006.

[3] Adams, C. and Lloyd, S., Understanding PKI: concepts, standards, and deployment considerations, Addison-Wesley Professional, 2003.

[4] NSA Suite B Cryptography, Suite B Implementers' Guide to NIST SP 800-56A, July 28, 2009. (https://www.nsa.gov/ia/_files/suiteb_implementer_g-113808.pdf, accessed 5 April 2016)

[5] Announcing the ADVANCED ENCRYPTION STANDARD (AES), NIST, 26 November 2001. (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, accessed 5 April 2016)

[6] Barker, E., Chen, L., Roginsky, A., and Smid, M., Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography, NIST, May 2013. (http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf, accessed 5 April 2016)

[7] AES Key Wrap Specification, NIST, 16 November 2001. (http://csrc.nist.gov/groups/ST/toolkit/documents/kms/key-wrap.pdf, accessed 5 April 2016)

[8] The relic-toolkit Open Source Project on Open Hub. (https://www.openhub.net/p/relic-toolkit, accessed 5 April 2016)

[9] St Denis, T., LibTomCrypt Developer Manual. (http://www.opensource.apple.com/source/CommonCrypto/CommonCrypto-55010/Source/libtomcrypt/doc/libTomCryptDoc.pdf, accessed 5 April 2016).

[10] X.509, Telecommunication Standardization Sector of ITU (ITU-T), October 2012. (https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-201210-I!!PDF-E&type=items, accessed 5 April 2016)

[11] Housley, R., Cryptographic Message Syntax (CMS), IETF Network Working Group, September 2009. (https://tools.ietf.org/html/rfc5652, accessed 5 April 2016)

[12] Introduction to ASN.1, ITU-T. (http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx, accessed 5 April 2016)

[13] Ambrose, J. et al., Power Analysis Side Channel Attacks: The Processor Design-level Context, Omniscriptum Gmbh & Company Kg., 2010.

[14] Kipmaa, H., Rogaway, P., and Wagner, D., Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption. (http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ctr/ctr-spec.pdf, assessed 5 April 2016)

[15] AMBA Specifications, ARM. (http://www.arm.com/products/system-ip/amba-specifications.php, accessed 5 April 2016)

[16] Vai, M., Nahill, B., Kramer, J., Geis, M., Utin, D., Whelihan, D., and Khazan, R., Secure Architecture for Embedded Systems, IEEE High Performance Extreme Computing Conference, September 2015.